

**Southwest University of Science and Technology**

**Data Structure**

**Design of Huffman Encoder/Decoder**

****

**NAME: AHMED MD FOYSAL**

**中文名：李艺恒**

**STUDENT ID: 4420190015**

**Major: Computer Science and Technology(undergraduate)**

#### **Design of Huffman Encoder/Decoder**

**H**uffman Coding is a technique of compressing data to reduce its size without losing any of the details. It was first developed by David Huffman. Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

Problem Description:

Using Huffman coding for communication can greatly improve channel utilization, shorten information transmission time, and reduce transmission costs. However, this requires pre-encoding the data to be transmitted through an encoding system at the sending end, and decoding (recovering) the transmitted data at the receiving end. For duplex channels (that is, channels that can transmit information in both directions), a complete encoding/decoding system is required at each end. Try to write a Huffman encoding/decoding system for such a messaging station.

A complete system shall have the following functions:  
(1) I: Initialization. Read in the size n of character set, n characters and N weights from the terminal, establish Huffman tree, and save it in the file hfmTree.  
(2) E: Encoding. Using the built Huffman tree (if not in memory, read it from the file hfmTree), the text in the file ToBeTran is encoded, and then the result is stored in the file CodeFile.  
(3) D: Decoding. The code in the CodeFile is decoded by using the Huffman tree, and the result is stored in the Textfile.

Here are the options:  
(4) P: Print code file. The CodeFile is displayed on the terminal in a compact format with 50 codes per line. At the same time, the encoding file in the form of this character is written to the file CodePrin.  
(5) T: print tree printing. Display the Huffman tree in memory in an intuitive way (such as tree) on the terminal, and write the Huffman tree in the form of this character into the file TreePrint.

#### Test requirements

(1) It is known that only eight characters may appear in the communication of a system, and their frequencies are 0.05, 0.29, 0.07, 0.08, 0.14, 0.23, 0.03 and 0.11 respectively. Try to design Huffman code.  
(2) The Huffman tree is established with the actual statistical data of character set and frequency given in the table below, and the coding and decoding of the following message is realized: "THIS PROGRAME IS MY FAVORITE".  
Character A B C D E F G H I J K L M  
Frequency 186 64 13 22 32 103 21 15 47 57 1 5 32 20  
Character N O P Q R S T U V W X Y Z   
Frequency: 57 63 15 1 48 51 80 23 8 18 1 16 1

#### Implementation tips

(1) The encoding results are stored as text in the file Codefile.  
(2) The user interface can be designed as a "menu" mode: display the above function symbols, plus "Q", indicating Quit. Please type a selection character. This menu will be displayed after this function is executed until "Q" is selected by a user.  
(3) During the execution of a program, after the first execution of I, D or C commands, the heffman tree is already in memory and does not need to be read in. I command is not necessarily executed in each execution, because the file hfmTree may have been built.

### Demand analysis

Data structure and algorithm is one of the important core courses of computer science and technology, which plays a very important role in the learning process of computer specialty. Course design of data structure and algorithm is to use the knowledge and technology of this course and related courses so far to solve practical problems. Especially in the application of non-numerical calculation, it is necessary to select the appropriate data structure and design an effective algorithm to meet certain time and space constraints.  
This course design requires students to complete a relatively complete application demand analysis independently. In the process of designing and compiling a program with a certain scale, deepen the understanding of the basic concepts, theories and methods in the course of data structure and algorithm; train the ability to comprehensively use the knowledge learned to deal with practical problems, strengthen the object-oriented programming concept; and improve the level of self-programming and debugging.

### Relationship Call

decoder function

encoder function

input weight function

initialization Huffman tree function

Definition of Structure

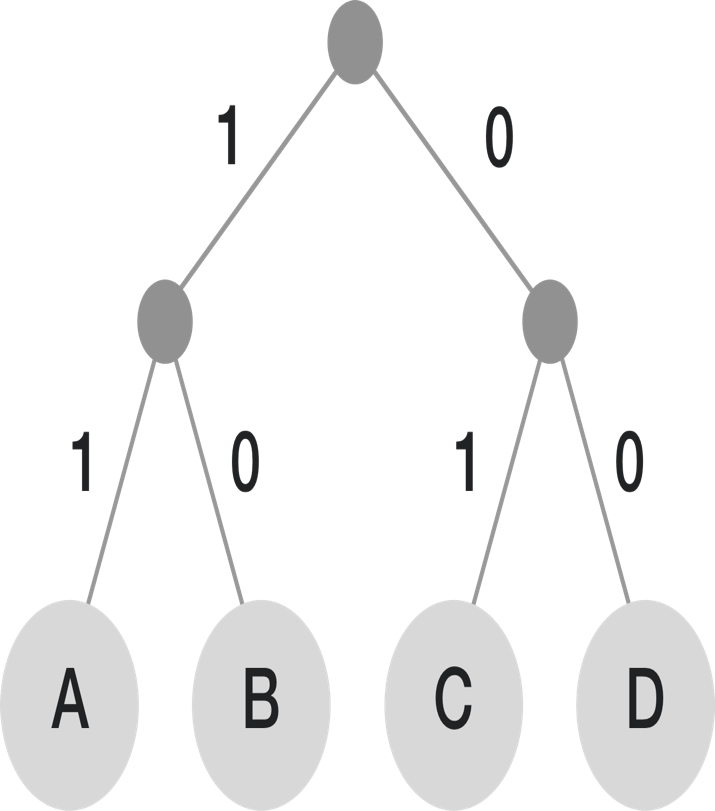
print tree function

Huffman Decoder/Encoder

### 

Construction flow chart of Huffman tree

### Huffman Initialization, Encode, Decode

**Huffman coding** is an efficient method of compressing data without losing information. In computer science, information is encoded as bits—1's and 0's. [Strings of bits](https://brilliant.org/wiki/binary-numbers/) encode the information that tells a computer which instructions to carry out. Video games, photographs, movies, and more are encoded as strings of bits in a computer. Computers execute billions of instructions per second, and a single video game can be billions of bits of data. It is easy to see why efficient and unambiguous information encoding is a topic of interest in computer science. Huffman coding provides an efficient, unambiguous code by analysing the frequencies that certain symbols appear in a message. Symbols that appear more often will be encoded as a shorter-bit string while symbols that aren't used as much will be encoded as longer strings. Since the frequencies of symbols vary across messages, there is no one Huffman coding that will work for all messages. This means that the Huffman coding for sending message X may differ from the Huffman coding used to send message Y. There is an algorithm for generating the Huffman coding for a given message based on the frequencies of symbols in that particular message. Huffman coding works by using a frequency-sorted [binary tree](https://brilliant.org/wiki/binary-search-trees/) to encode symbols. Huffman tree generated from the exact frequencies of the text "this is an example of a Huffman tree". The frequencies and codes of each character are below. Encoding the sentence with this code requires 135 (or 147) bits, as opposed to 288 (or 180) bits if 36 characters of 8 (or 5) bits were used. (This assumes that the code tree structure is known to the decoder and thus does not need to be counted as part of the transmitted information.)

Visualisation of the use of Huffman coding to encode the message "A\_DEAD\_DAD\_CEDED\_A\_BAD\_BABE\_A\_BEADED\_ABACA\_BED". In steps 2 to 6, the letters are sorted by increasing frequency, and the least frequent two at each step are combined and reinserted into the list, and a partial tree is constructed. The final tree in step 6 is traversed to generate the dictionary in step 7. Step 8 uses it to encode the message.

Outline design

(1) typedef struct{  
int weight;  
char Data; / / the character holding the node  
int Parent, Lchild, Rchild;  
}HTNode,\*HuffmanTree;

(2) typedef char\*\* HuffmanCode;  
(3) void HuffmanCoding(HuffmanTree &,char \*,int \*,int);  
(4) Void select (humantree HT, int j, int \* S1, int \* S2); / / select the two nodes whose parents are 0 and whose weights are the least  
(5) void Initialization(); / / initialize Huffman tree  
(6) void Coding(); / / Huffman coding  
(7) void Decoding(); / / Huffman decoding  
(8) void find(HuffmanTree &HT,char \*code,char \*text,int i,int m);  
(9) HuffmanTree HT;  
(10) int n=0; / / number of leaf nodes of Huffman tree  
(11) int main() / / main function

### Procedure description

This program is a Huffman encoder / decoder. It uses Initialization function to initialize Huffman tree, Coding function to Huffman Coding and decoding function to Huffman decoding. The Coding function involves the select function. The purpose of the select function is to find two nodes whose parents are 0 and whose weights are the least. The main function of the find function is to find the character corresponding to the Huffman code in the Huffman tree recursively and write it into Textfile.txt.

Detailed design (Code):

//

//  main.cpp

//  xyz

//

//  Created by AHMED MD FOYSAL (李艺恒） 4420190015  on 2020/12/28.

//

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

**typedef** **struct**{

**int** weight;

**char** Data;                                     //Characters to hold nodes

**int** Parent,Lchild,Rchild;

}HTNode,\*HuffmanTree;

**typedef** **char**\*\* HuffmanCode;

**void** HuffmanCoding(HuffmanTree &,**char** \*,**int** \*,**int**);

**void** select(HuffmanTree HT,**int** j,**int** \*s1,**int** \*s2);   //Select two nodes with zero parents and minimum weight

**void** Initialization();                                            // Initialize heffman tree

**void** Coding();                                                         //Huffman code

**void** Decoding();                                                     //Huffman decoding

**void** find(HuffmanTree &HT,**char** \*code,**char** \*text,**int** i,**int** m);

HuffmanTree HT;

**int** n=0;                                                   //Number of leaf nodes of heffman tree

//------------------------------------------------------------------------------

                                                                        //Main function

**int**    main()

{

**char** T;

**while** (1)

    {

        printf("\n");

        printf("      \*\*============================================================\*\*\n");

        printf("    \*\*\*\*=                                                          =\*\*\*\*\n");

        printf("  \*\*\*\*\*\*=             Design of Huffman Encoder/Decoder            =\*\*\*\*\*\*\n");

        printf("    \*\*\*\*=                                                          =\*\*\*\*\n");

        printf("      \*\*============================================================\*\*\n");

        printf("        ============================================================\n");

        printf("        =                  Huffman Encoder/Decoder                 =\n");

        printf("        ============================================================\n");

        printf("        =   Initialization     -----------  I                      =\n");

        printf("        =   Encoding           -----------  C                      =\n");

        printf("        =   Decoding           -----------  D                      =\n");

        printf("        =   Exit               -----------  Q                      =\n");

        printf("        ============================================================\n");

        printf("        =   Please input ( I C D Q) :                              =\n");

        printf("        ------------------------------------------------------------\n");

        scanf("%c",&T);

**switch**(T)

        {

**case** 'I':

                Initialization();

**break** ;

**case** 'C':

                Coding();

**break** ;

**case** 'D':

                Decoding();

**break** ;

**case** 'Q':

                exit(1);

**default**        :

                printf(    "\t\t\t\t Input error \n" );

        }

        getchar();

    }

**return**    0;

}

//------------------------------------------------------------------------------

                   //Initialization function, input n characters and their corresponding weights, and establish heffman tree according to the weights

**void** Initialization()

{

    FILE \*fp;

**int**    i,w[52];                                                //Weight of stored characters

**char** character[52];                                             // Store n characters

    printf("\t\t\t\t Please enter the number of characters:");

    scanf("%d",&n);

    printf("\t\t\t\t Please input%d Characters and weight:\n\t\t\t\t" ,n);

**for**(i=0;i<n;i++)

    {

**char**  b=getchar();

        scanf("%c" ,&character[i]);

        scanf("%d",&w[i]);                                 // Input characters and corresponding weights

    }

    HuffmanCoding(HT,character,w,n);                                 // Establish heffman tree

**if** ((fp=fopen("/Users/ahmedmdfoysal/Desktop/Huffman Testcpp01/Huffman Testcpp01/hfmTree.txt","w"))==**NULL**)

        printf("\t\t\t\thfmTree.txt    Open failure \n"  );

**for**(i=1;i<=2\*n-1;i++)

    {

**if**(fwrite(&(HT[i].Data),**sizeof** (**char**),1,fp)!=1)

            printf(    "\t\t\t\t fail to write to file  \n");

**if**(fwrite(&(HT[i].weight),**sizeof**(**int**),1,fp)!=1)

            printf(    "\t\t\t\t fail to write to file  \n");

**if**(fwrite(&(HT[i].Parent),**sizeof**(**int**),1,fp)!=1)

            printf(    "\t\t\t\t fail to write to file  \n");

**if**(fwrite(&(HT[i].Lchild),**sizeof**(**int**),1,fp)!=1)

            printf(    "\t\t\t\t fail to write to file  \n");

**if**(fwrite(&(HT[i].Rchild),**sizeof**(**int**),1,fp)!=1)

            printf(    "\t\t\t\t fail to write to file  \n");

    }

    printf("\t\t\t\t Huffman tree has been established and stored in hfmTree.txt\n");

    fclose(fp);

}

//------------------------------------------------------------------------------

                                                            //An algorithm for constructing heffman tree

**void** HuffmanCoding(HuffmanTree &HT,**char** \*character, **int** \*w, **int** n)

{

**int**    m,i,s1,s2;

    HuffmanTree p;

**if**(n<=1)

**return**  ;

    m=2\*n-1;

    HT=(HuffmanTree)malloc((m+1)\* **sizeof**(HTNode));

**for**(p=HT+1,i=1;i<=n;++i,++p,++character,++w)                       // Initial value

    {

        p->Data=\*character;

        p->weight=\*w;

        p->Parent=0;

        p->Lchild=0;

        p->Rchild=0;

    }

**for**(;i<=m;++i,++p)                                   // Assign an initial value of 0 to subsequent nodes

    {

        p->Data=0;

        p->weight=0;

        p->Parent=0;

        p->Lchild=0;

        p->Rchild=0;

    }

**for**(i=n+1;i<=m;++i)                                           // Generate new node

    {

        select(HT,i-1,&s1,&s2);

        HT[s1].Parent=i;HT[s2].Parent=i;

        HT[i].Lchild=s1;HT[i].Rchild=s2;

        HT[i].weight=HT[s1].weight+HT[s2].weight;

    }

}

//------------------------------------------------------------------------------

                                              //Select two nodes with zero parents and minimum weight

**void** select(HuffmanTree HT,**int** j, **int** \*s1, **int** \*s2)

{

**int**    i;

**for**(i=1;i<=j;i++)

**if** (HT[i].Parent==0)

        {

            \*s1=i;

**break**;

        }

**for**(;i<=j;i++)

**if**((HT[i].Parent==0)&&(HT[i].weight<HT[\*s1].weight))

        \*s1=i;                                    // Then s1 is the sequence number of the smallest node

    HT[\*s1].Parent=1;    // Assign a value of 1 to HT[\*s1].Parent in advance, so as to avoid being affected when finding summary points and judging conditions

**for**(i=1;i<=j;i++)

**if**(HT[i].Parent==0)

        {

            \*s2=i;

**break**;

        }                        // There's a cycle for those who haven't been visited, which is the starting point for comparison

**for**(;i<=j;i++)

**if**((HT[i].Parent==0)&&(i!=\*s1)&&(HT[i].weight<HT[\*s2].weight))

        \*s2=i;                                      // Find the node with less weight

}

//------------------------------------------------------------------------------

//Carry out Huffman coding

**void** Coding()

{

    FILE \*fp,\*fw;

**int**    i,f,c,r,start;

**char** \*cd;

**char** temp;

    HuffmanCode HC;

    {

        HC=(HuffmanCode)malloc((n+1)\* **sizeof** (**char**\*));

        cd=(**char** \*)malloc(n\***sizeof**(**char**));

        cd[n-1]='\0';

**for**(i=1;i<=n;++i)

        {

            start=n-1;

**for**(c=i,f=HT[i].Parent;f!=0;c=f,f=HT[f].Parent)

**if** (HT[f].Lchild==c)

                    cd[--start]='0';

**else**

                    cd[--start]='1';

            HC[i]=(**char** \*)malloc((n-start)\* **sizeof**(**char**));

            strcpy(HC[i],&cd[start]);

        }

        free(cd);

    }

**if**((fp=fopen("/Users/ahmedmdfoysal/Desktop/Huffman Testcpp01/Huffman Testcpp01/ToBeTran.txt","rb"))==**NULL**)

            printf("\t\t\t\tToBeTran.txt Open failure \n");

**if**((fw=fopen("CodeFile.txt","wb+"))==**NULL**)      // Create new file for read / write

            printf("\t\t\t\tCodeFile.txt Open failure \n");

        fscanf(fp,"%c",&temp);    // Read the first character into the temp variable from the file referred to by fp

**while**(!feof(fp))

        {

**for**(i=1;i<=n;i++)

**if** (HT[i].Data==temp)

**break** ;                           // Jump out of loop if same as node 1

**for**(r=0;HC[i][r]!='\0';r++)    // Find the location of the character in the Huffman tree

                fputc(HC[i][r],fw);    // Save the code corresponding to the character into the file, and write HC[i][r] to the file indicated by fw

            fscanf(fp,"%c",&temp);                     // Read next character from file

        }

        fclose(fw);

        fclose(fp);

        printf("\n\t\t\t\thfmTree.txt    Encoded successfully, saved in CodeFile.txt\n"    );

}

//------------------------------------------------------------------------------

//Perform heffman decoding

**void** Decoding()

{

    FILE \*fp,\*fw;

**int**    m,i;

**char** \*code,\*text,\*p;

**if**((fp=fopen("/Users/ahmedmdfoysal/Desktop/Huffman Testcpp01/Huffman Testcpp01/CodeFile.txt","rb"))==**NULL**)                  // Open file for read-only

        printf("\t\t\t\tCodeFile.txt    Open failure \n");

**if**((fw=fopen("Textfile.txt","wb+"))==**NULL**) // Create new file read / write, if created, the content will be cleared

        printf(    "\t\t\t\tTextfile.txt    Open failure \n");

    code=(**char**\*)malloc(**sizeof**(**char**));

    fscanf(fp,"%c",code);                             // Read in a 0 or 1 character from a file

**for**(i=1;!feof(fp);i++)

    {

        code=(**char** \*)realloc(code,(i+1)\***sizeof**(**char**));

        fscanf(fp,"%c",&code[i]);                 // Read next 0 or 1 character from file

    }

    //    So far, 0 and 1 in CodeFile.txt have been read in and stored in code array

    text=(**char** \*)malloc(100\***sizeof**(**char**));

    p=text;

    m=2\*n-1;

**if**(\*code=='0')

        find(HT,code,text,HT[m].Lchild,m);                   // Find from the left subtree of the root node

**else**

        find(HT,code,text,HT[m].Rchild,m);                   // Find from the right subtree of the root node

**for**(i=0;p[i]!='\0';i++)               // Store the decoded characters in Textfile.txt

        fputc(p[i],fw);                               // Write p[i] to the file indicated by fw

    fclose(fp);

    fclose(fw);

    printf(    "\n\t\t\t\tCodeFile.txt    Decoding successful, stored in Textfile.txt\n"    );

}

//------------------------------------------------------------------------------

//A recursive algorithm for finding the corresponding leaf node according to 01 string in decoding

**void** find(HuffmanTree &HT,**char** \*code,**char** \*text,**int** i, **int** m)

{

**if**(\*code!='\0')                                             // If decoding is not finished

    {

        code++;

**if**(HT[i].Lchild==0&&HT[i].Rchild==0)                   // If a leaf node is found

        {

            \*text=HT[i].Data;                           // Save leaf node characters in text

            text++;

**if**((\*code=='0'))

                find(HT,code,text,HT[m].Lchild,m);       // Continue to find from the left subtree of the root node

**else**

                find(HT,code,text,HT[m].Rchild,m);       // Continue to search from the right subtree of the root node

        }

**else**

**if**(\*code=='0')

                find(HT,code,text,HT[i].Lchild,m);              // Find from the left subtree of this node

**else**

                find(HT,code,text,HT[i].Rchild,m);             // Find from the right subtree of this node

    }

**else**

        \*text='\0';

}

### Debugging analysis

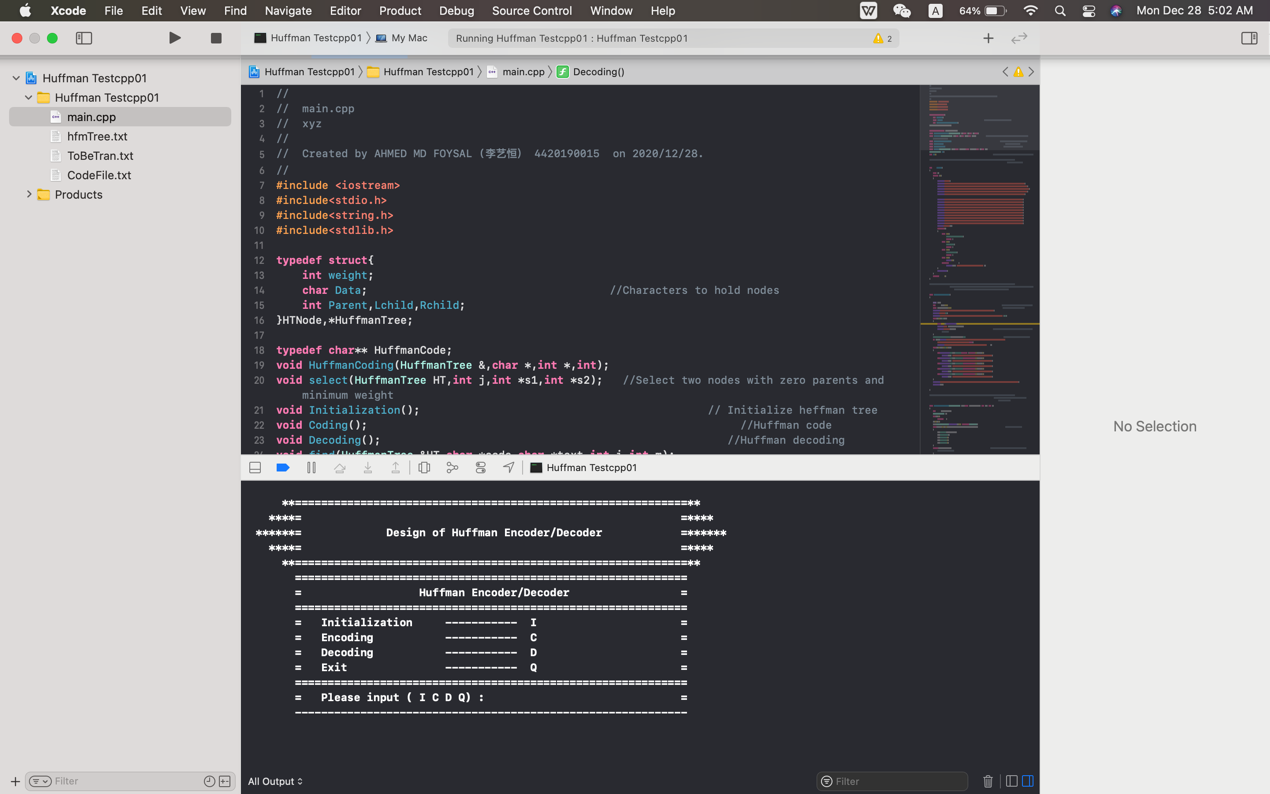


Fig: Debugging in Xcode

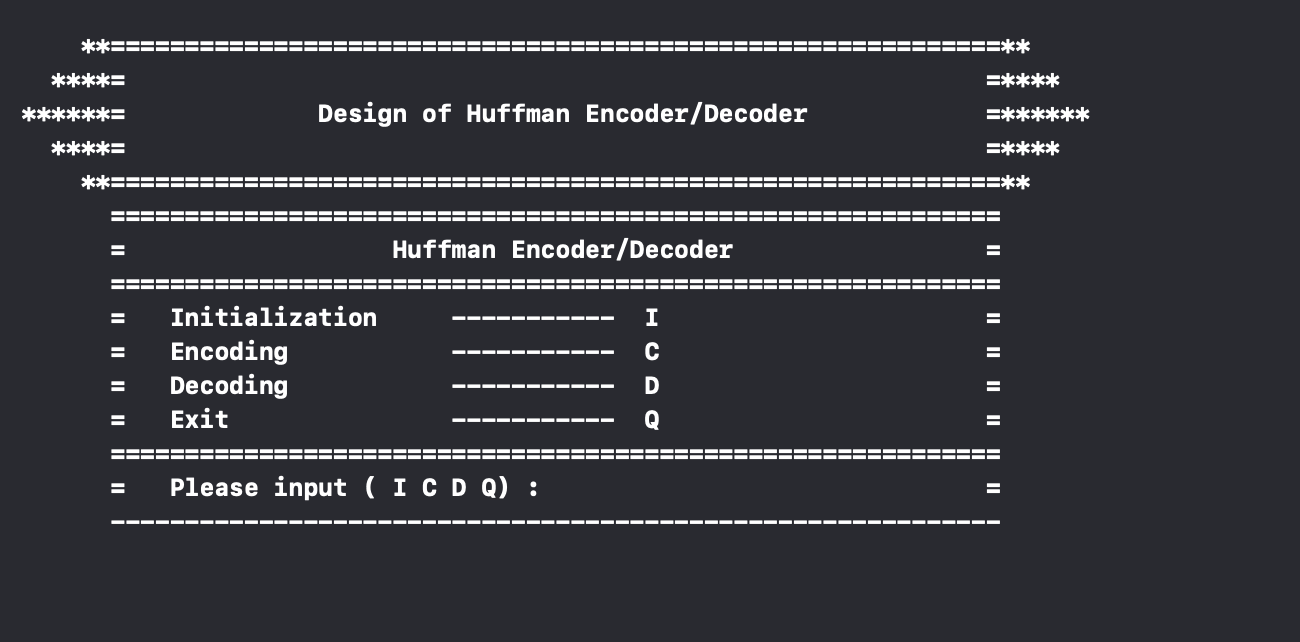
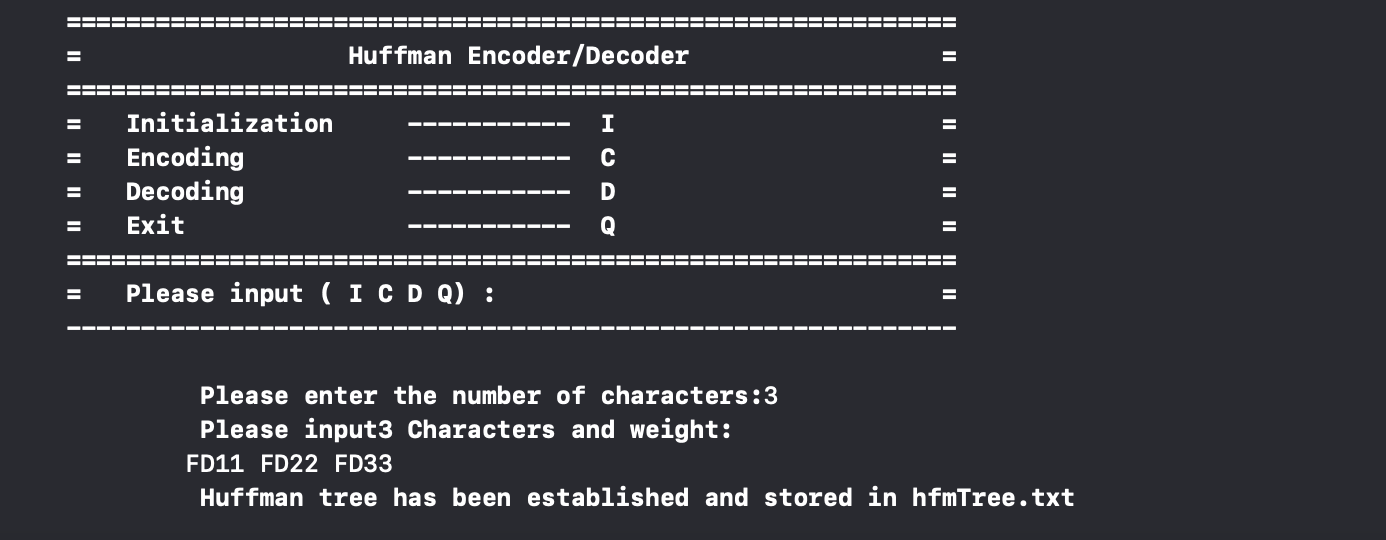
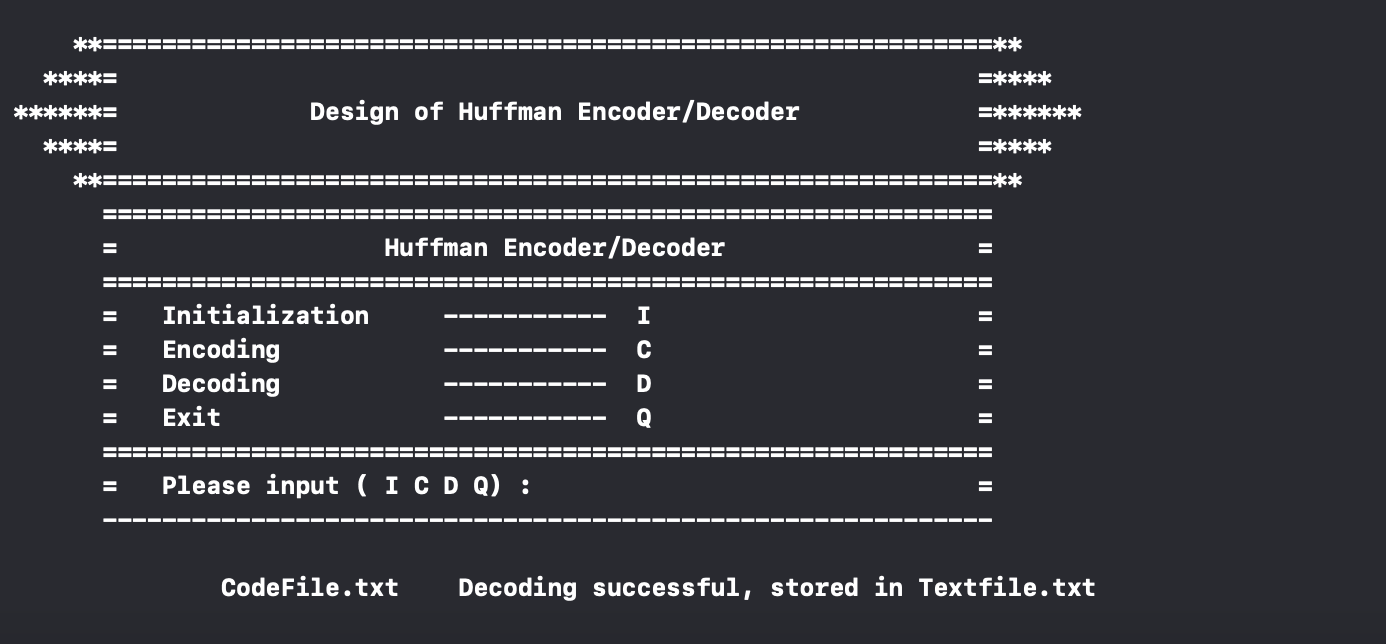


Fig: Menu

 Fig: Input Weight



### Time Complexity Analysis

Since Huffman coding uses min Heap data structure for implementing priority queue, the complexity is O(nlogn). This can be explained as follows-

* Building a min heap takes O(nlogn) time (Moving an element from root to leaf node requires O(logn) comparisons and this is done for n/2 elements, in the worst case).
* Building a min heap takes O(nlogn) time (Moving an element from root to leaf node requires O(logn) comparisons and this is done for n/2 elements, in the worst case).

Since building a min heap and sorting it are executed in sequence, the algorithmic complexity of entire process computes to O(nlogn)

We can have a linear time algorithm as well, if the characters are already sorted according to their frequencies.

|  |  |
| --- | --- |
| Huffman Encoding | |
| Advantage | Disadvantage |
| * This encoding scheme results in saving lot of storage space, since the binary codes generated are variable in length * It generates shorter binary codes for encoding symbols/characters that appear more frequently in the input string * The binary codes generated are prefix-free | * Lossless data encoding schemes, like Huffman encoding, achieve a lower compression ratio compared to lossy encoding techniques. Thus, lossless techniques like Huffman encoding are suitable only for encoding text and program files and are unsuitable for encoding digital images. * Huffman encoding is a relatively slower process since it uses two passes- one for building the statistical model and another for encoding. Thus, the lossless techniques that use Huffman encoding are considerably slower than others. * Since length of all the binary codes is different, it becomes difficult for the decoding software to detect whether the encoded data is corrupt. This can result in an incorrect decoding and subsequently, a wrong output. |

Huffman Encoding Advantage and Disadvantage

|  |
| --- |
| Real-life applications of Huffman Encoding-  * Huffman encoding is widely used in compression formats like GZIP, PKZIP (winzip) and BZIP2. * Multimedia codecs like JPEG, PNG and MP3 uses Huffman encoding (to be more precised the prefix codes) * Huffman encoding still dominates the compression industry since newer arithmetic and range coding schemes are avoided due to their patent issues |